

Exemple de primitives logo implémentées

- https://turing.cs.hbg.psu.edu/LOCAL/ucblogo/ucbl_7.htm
- <https://fmslogo.sourceforge.io/manual/workspace-control.html>
- <https://www.calormen.com/jslogo/language.html>

The section numbers are per the [Berkeley Logo Reference Manual](#) in Brian Harvey's *Computer Science Logo Style volume 2: Advanced Techniques* for ease of comparison.

1. [Data Types and Syntax](#)
2. [Data Structure Primitives](#)
3. [Communication](#)
4. [Arithmetic](#)
5. [Logical Operations](#)
6. [Graphics](#)
7. [Workspace Management](#)
8. [Control Structures](#)

Data Types and Syntax

`"word`
`'word`
`number`

Word. (Quoted words are terminated by `[](){}|` or whitespace, `\` to escape.)

```
show "hello
show "hello\ logo
show 12.34
```

`:variable`

Input definition/variable reference

```
show :name
```

`[word ...]`

List of words or lists, whitespace separated

```
show [1 2 3]
```

```
{ word ... }
```

```
{ word ... }@origin
```

Array of words, whitespace separated

```
show {1 2 3}
```

```
( expression )
```

Parenthesis can be used to group expressions

```
show ( 1 + 2 ) * 3
```

```
procedure input ...
```

Call procedure with default number of inputs

```
show "hello
```

```
( procedure input ... )
```

Call procedure with an arbitrary number of inputs

```
(show "hello :name)
```

2. Data Structure Primitives

2.1 Constructors

```
word expr expr
```

```
(word expr ...)
```

Concatenate two (or more) words into one word

```
show (word "a "b "c)
```

```
list thing1 thing2
```

```
(list thing1 thing2 ...)
```

Create a new list from the inputs

```
show (list 1+2 2+3 3+4)
```

```
sentence thing1 thing2
```

```
(sentence thing1 thing2 ...)
```

```
se thing1 thing2
```

```
(se thing1 thing2 ...)
```

Create a new list from the inputs (if non-lists) or members of the inputs (if lists)

```
show sentence [1 2 3] [and so on]
```

```
fput thing list
```

```
lput thing list
```

Outputs list, but with thing added as a new first/last item; if list is a word, concatenates

```
show fput 0 [ 1 2 3 ]
```

```
show fput "a "bcd
```

```
show lput 4 [ 1 2 3 ]
```

```
show lput "e "bcd
```

```
array size
```

```
(array size origin)
```

Create a new array. The default origin is 1.

```
show array 10
```

```
mdarray [dimensions ...]
```

```
(mdarray [dimensions ...] origin)
```

Create a new multi-dimensional array. The default origin is 1.

```
show mdarray [2 2]
```

`listtoarray list`

`(listtoarray list origin)`

Create a new array populated by members of a list

```
show listtoarray [ 1 3 5 7 ]
```

`arraytolist array`

Create a new list populated by members of a array

```
show arraytolist { 2 4 6 8 }
```

`combine thing1 thing2`

If thing2 is a word, like `word`; otherwise, like `fput`

```
show combine "a [b c]
```

`reverse list`

Outputs a list with the items in reverse order; if input is a word, reverses characters

```
show reverse [ 1 2 3 ]
```

```
show reverse "abc"
```

`gensym`

Outputs a unique string, e.g. G1, G2, etc.

```
show gensym
```

2.2 Data Selectors

`first list`

`last list`

Outputs the first or last item (or character) from the list (or word), respectively

```
show first [ 1 2 3 ]
```

```
show first "abc"
```

```
show last [ 1 2 3 ]
```

```
show last "abc"
```

`firsts listoflists`

Outputs a list of the first item (or character) of each sublist (or word)

```
show firsts [ [1 2 3] [a b c] ]
```

```
show firsts [ abc def ]
```

`butfirst list`

`bf list`

`butlast list`

`bl list`

Outputs the list (or word), except for the first/last item (or character)

```
show butfirst [ 1 2 3 ]
```

```
show butfirst "abc"
```

```
show butlast [ 1 2 3 ]
```

```
show butlast "abc"
```

`butfirsts listoflists`

`bfs listoflists`

Outputs a list of sublists (or words) without the first item (or characters)

```
show butfirsts [ [ 1 2 3 ] [ a b c ] ]
```

```
show butfirsts [ abc def ]
```

`item index thing`

Outputs the indexth item of the list or array or word

```
show item 2 [ 1 2 3 ]
```

```
show item 2 "abc
```

```
mditem [index ...] thing
```

Outputs an item from a multi-dimensional array

```
show mditem [2 1] {{a b} {c d}}
```

```
pick list
```

Outputs one item from a list (or word), at random

```
show pick [ 1 2 3 ]
```

```
show pick "abc
```

```
remove thing list
```

Outputs the list (or word) with any occurrences of thing removed

```
show remove "b [ a b c ]
```

```
show remove "a "banana
```

```
remdup list
```

Outputs the list (or word) with duplicates removed

```
show remdup [ 1 2 3 2 3 4 3 4 5 ]
```

```
show remdup "banana
```

```
quoted thing
```

Outputs thing with " prepended if a word, or just thing otherwise.

```
show quoted "abc
```

```
split thing list
```

Outputs the list split into a list of lists (or list of words) wherever thing appears.

```
show split "a "banana
```

```
show split 3 [1 2 3 4 1 2 3 4]
```

2.3 Data Mutators

```
setitem index array value
```

Sets the *index*th item of the array to *value* (circular arrays prevented)

```
make "a { 1 2 3 } setitem 2 :a "x show :a
```

```
mdsetitem [index...] array value
```

Sets an item in a multi-dimensional array to *value* (circular arrays prevented)

```
make "a {{1 2} {3 4}} mdsetitem [2 1] :a "x show :a
```

```
.setfirst list value
```

Sets the first item of the list to *value*

```
make "a [ 1 2 3 ] .setfirst :a "7 show :a
```

```
.setbf list value
```

Sets the 'butfirst' of the list to the items in *value* (a list)

```
make "a [ 1 2 3 ] .setbf :a [ 4 9 ] show :a
```

```
.setitem index array value
```

Sets the *index*th item of the array to *value* (circular arrays allowed)

```
make "a { 1 2 3 } setitem 2 :a "x show :a
```

```
push stackname thing
```

```
pop stackname
```

Push to/pop from a stack i.e. list variable. Stacks grow from the front. Works on words.

```
make "s [ 2 1 ] push "s 3 show :s show pop "s
```

```
make "s "ba push "s "c show :s show pop "s
```

```
queue stackname thing
```

```
dequeue stackname
```

Add/remove from a queue i.e. list variable, Queues grow from the end. Works on words.

```
make "q [ 1 2 ] queue "q 3 show :q show dequeue "q
```

```
make "q "ab queue "q "c show :q show dequeue "q
```

2.4 Predicates

Predicates return 1 (true) or 0 (false)

```
wordp thing
```

```
word? thing
```

```
listp thing
```

```
list? thing
```

```
arrayp thing
```

```
array? thing
```

```
numberp thing
```

```
number? thing
```

Test if thing is a word, list, array, or number respectively.

```
emptyp expr
```

```
empty? expr
```

Test if thing is an empty list or empty string.

```
equalp expr expr
```

```
equal? expr expr
```

```
expr = expr
```

```
notequalp expr expr
```

```
notequal? expr expr
```

```
expr <> expr
```

Equality/inequality tests. Compares strings, numbers, or lists (equal if length and all members are equal).

```
beforep thing1 thing2
```

```
before? thing1 thing2
```

Test string collation order.

```
.eq thing1 thing2
```

Test if things have the same identity.

```
memberp thing list
```

```
member? thing list
```

Test if thing is equal to any member of list.

```
substringp thing1 thing2
```

```
substring? thing1 thing2
```

Test if thing1 is a substring of thing2.

2.5 Queries

```
count thing
```

Outputs length of a list or number of characters in a string

```
show count [ 1 2 3 ]
```

```
show count "hello"
```

```
ascii expr
```

Outputs ASCII (actually, Unicode) code point for first character of string

```
show ascii "abc"
```

`char expr`

Outputs Unicode character at specified code point

`show char 97`

`member thing list`

Outputs the list (or word) from the first occurrence of thing to the end, or empty list (or word)

`show member "a "banana`

`show member 2 [1 2 3 4]`

`uppercase expr`

`lowercase expr`

Outputs string converted to upper/lowercase

`show uppercase "abc`

`show lowercase "ABC`

`standout expr`

Outputs string with alphanumeric characters in bold

`show standout "ABCabc123`

`parse word`

Outputs word parsed as a list.

`show parse "1+2`

`runparse word`

Outputs word parsed as instructions.

`show runparse "1+2`

3. Communication

3.1 Transmitters

`print thing`

`pr thing`

`(print thing1 thing2 ...)`

`(pr thing1 thing2 ...)`

Print inputs to the text screen, separated by spaces, and followed by a newline. Square brackets are only put around sublists.

`print "hello`

`type thing`

`(type thing1 thing2 ...)`

Like `print` but with no trailing newline.

`type "hel type "lo`

`show thing`

`(show thing1 thing2 ...)`

Like `print` but with square brackets around list inputs.

`show "hello`

3.2 Receivers

`readlist`

`(readlist promptstr)`

Prompt the user for a line of input. The result is a list of words.

show readlist

make "colors (readlist [Type some colors:]) show :colors

readword

(readword *promptstr*)

Prompt the user for a line of input. The result (including spaces) is the single word output.

show readword

make "name (readword [What is your name?]) show :name

3.4 Terminal Access

cleartext

ct

Clear the text screen.

settextcolor *color*

Change the text color.

textcolor

Output the current text color.

increasefont

decreasefont

Increase/decrease the text size.

settextsize *height*

Change the text size (in pixels).

textsize

Output the current text size (in pixels).

setfont *name*

Change the text font.

font

Output the current text font.

4. Arithmetic

4.1 Numeric Operations

Inputs are numbers or numeric expressions, output is a number.

sum *expr expr*

(sum *expr ...*)

expr + *expr*

difference *expr expr*

expr - *expr*

product *expr expr*

(product *expr ...*)

expr * *expr*

quotient *expr expr*

(quotient *expr*)

expr / *expr*

power *expr expr*

`expr ^ expr`

Add, subtract, multiply, divide, and raise-to-the-power-of respectively. A single input to quotient returns the reciprocal.

`remainder expr expr`

`expr % expr`

`modulo expr expr`

Outputs the remainder (modulus). For `remainder` and `%` the result has the same sign as the first input; for `modulo` the result has the same sign as a the second input.

`minus expr`

`- expr`

Unary minus sign must begin a top-level expression, follow an infix operator, or have a leading space and no trailing space.

`abs num`

Absolute value

`int num`

`round num`

Truncate or round a number, respectively.

`sqrt expr`

`exp expr`

`log10 expr`

`ln expr`

Square root, e to the power of, common logarithm, and natural logarithm, respectively.

`arctan expr`

`(arctan x y)`

`sin expr`

`cos expr`

`tan expr`

The usual trig functions. Angles are in degrees.

`radarctan expr`

`(radarctan x y)`

`radsin expr`

`radcos expr`

`radtan expr`

The usual trig functions. Angles are in radians.

`iseq first last`

Outputs a list with integers from *first* to *last*, inclusive

show `iseq 1 10`

`rseq first last count`

Outputs a list of *count* numbers from *first* to *last*, inclusive

show `rseq 1 9 5`

4.2 Numeric Predicates

`lessp expr expr`

`less? expr expr`

`expr < expr`

`greaterp expr expr`

`greater? expr expr`

`expr > expr`

`lessequalp expr expr`

`lessequal? expr expr`

```
expr <= expr
```

```
greaterqualp expr expr
```

```
greaterqual? expr expr
```

```
expr >= expr
```

Less than, greater than, less than or equal to, greater than or equal to, respectively. Inputs are numbers or numeric expressions, output is 1 (true) or 0 (false).

4.3 Random Numbers

```
random expr
```

```
(random start end)
```

Outputs a random number from 0 through one less than *expr*, or from start to end inclusive.

```
show random 10
```

```
show (random 1 6)
```

```
rerandom
```

```
(rerandom expr)
```

Reseeds the random number generator, either to a fixed value or the specified seed.

4.4 Print Formatting

```
form expr width precision
```

Outputs a formatted string with the result of a numeric expression with *precision* decimal places and padded on the left with spaces (if necessary) to be at least *width characters long*.

```
show form 1/3 10 3
```

4.5 Bitwise Operations

```
bitand expr expr
```

```
(bitand expr ...)
```

```
bitor expr expr
```

```
(bitor expr ...)
```

```
bitxor expr expr
```

```
(bitxor expr ...)
```

```
bitnot expr
```

Bitwise and, or, exclusive-or, and not, respectively.

```
ashift expr bitcount
```

Arithmetic bit shift. If bitcount is negative, shifts to the right, preserving sign.

```
lshift expr bitcount
```

Logical bit shift. If bitcount is negative, shifts to the right, zero-filling.

5. Logical Operations

```
true
```

Outputs 1

```
false
```

Outputs 0

```
and expr expr  
(and expr ...)  
or expr expr  
(or expr ...)  
xor expr expr  
(xor expr ...)  
not expr
```

Logical "and", "or", "exclusive-or", and "not", respectively. Inputs are numbers or numeric expressions, output is 1 (true) or 0 (false).

6. Graphics

An introduction to [Turtle Geometry](#).

6.1 Turtle Motion

```
forward expr  
fd expr
```

Move turtle forward *expr* pixels

```
fd 100
```

```
back expr  
bk expr
```

Move turtle backward *expr* pixels

```
bk 100
```

```
left expr  
lt expr
```

Rotate *expr* degrees counterclockwise

```
lt 90
```

```
right expr  
rt expr
```

Rotate *expr* degrees clockwise

```
rt 90
```

```
setpos [ expr expr ]  
setxy expr expr  
setx expr  
sety expr
```

Move turtle to the specified location

```
setpos [ 100 -100 ]
```

```
setxy -100 100
```

```
setheading expr  
seth expr
```

Rotate the turtle to the specified heading

```
setheading 45
```

```
home
```

Moves the turtle to center, pointing upwards

```
arc angle radius
```

Without moving the turtle, draws an arc centered on the turtle, starting at the turtle's heading.

arc 180 100

6.2 Turtle Motion Queries

pos

xcor

ycor

Outputs the current turtle position as [x y], x or y respectively

show pos

heading

Outputs the current turtle heading

show heading

towards [*expr expr*]

Outputs the heading towards the specified [x y] coordinates

show towards [0 0]

scrunch

Outputs the current graphics scaling factors

show scrunch

bounds

Outputs the current graphics screen bounds [xmin xmax ymin ymax]

show bounds

6.3 Turtle and Window Control

showturtle

st

Show the turtle

hideturtle

ht

Hide the turtle

clean

Clear the drawing area

clearscreen

cs

Same as clean and home together

wrap

If the turtle moves off the edge of the screen it will continue on the other side. (default)

window

The turtle can move past the edges of the screen, unbounded.

fence

If the turtle attempts to move past the edge of the screen it will stop.

fill

Does a paint bucket flood fill at the turtle's position.

arc 360 100 fill

filled *fillcolor* [*statements ...*]

Execute *statements* without drawing but keeping track of turtle movements. When complete, fill the region traced by the turtle with *fillcolor* and outline the region with the current pen style.

filled "red [repeat 5 [fd 100 rt 144]]

label *expr*

Draw a word (same logic as `print`) on the graphics display at the turtle location

repeat 8 [label "Logo rt 45]

setlabelheight *expr*

Set the height for text drawn by `label`, in pixels

setlabelheight 100 label "Logo

setlabelfont *expr*

Set the font for text drawn by `label`

setlabelfont "Times\ New\ Roman label "Logo

setscrunch *sx sy*

Set the graphics scaling factors

setscrunch 1 2 arc 360 100

setturtle *index*

Switch to the turtle numbered *index* (starting from 1 for the default turtle present at start). If the turtle has not been used yet, it will be created at the center, facing upwards, visible, with the pen down.

setturtle 2 rt 90 fd 100

ask *turtleindex* [*statements ...*]

Execute *statements* as turtle number *turtleindex*.

ask 2 [rt 90 fd 100]

clearturtles

Remove all turtles, keeping the current one as index 1.

fd 50 setturtle 2 rt 90 fd 100 clearturtles

6.4 Turtle and Window Queries

shownp

shown?

Outputs 1 if the turtle is shown, 0 if the turtle is hidden

turtlemode

Outputs `WRAP`, `WINDOW` or `FENCE`

labelsize

Outputs the height of text drawn by `label`, in pixels

labelfont

Outputs the name of the font drawn by `label`

turtle

Outputs the index of the currently-active turtle

turtles

Outputs the largest index that has been passed to `setturtle`

6.5 Pen and Background Control

pendown

pd

Turtle resumes leaving a trail

penup

pu

Turtle stops leaving a trail

```
penpaint  
ppt  
penerase  
pe  
penreverse  
px
```

Change the turtle drawing mode - *paint* (the default) leaves a colored trail, *erase* restores the background, *reverse* inverts the background.

```
setpw 10 px repeat 5 [ fd 100 rt 144 ]
```

```
setpencolor logocolor  
setpencolor csscolor  
setpencolor [ r g b ]
```

Set pen/text color. Color can be a standard Logo color number (0-15), CSS color string ([CSS color names or #rrggbb](#)), or in the list version, r/g/b values in 0...99.

The standard Logo colors are:

0 black	1 blue	2 green	3 cyan
4 red	5 magenta	6 yellow	7 white
8 brown	9 tan	10 green	11 aqua
12 salmon	13 purple	14 orange	15 gray

```
setpencolor 4  
setpencolor "red  
setpencolor "#ff0000  
setpencolor [ 99 0 0 ]  
setpalette colornumber csscolor  
setpalette colornumber [r g b]
```

Change one of the standard color entries (8 or above) to the given color.

```
setpalette 8 "pink setbg 8  
setpalette 8 "#ff4f00 setbg 8  
setpalette 8 [ 99 31 0 ] setbg 8
```

```
setpensize expr
```

Set pen width in pixels. If *expr* is a list, the first member is used.

```
setbackground color  
setscreencolor color  
setsc color
```

Set the background color; same options as `setpencolor`
setbackground "red

6.6 Pen Queries

```
pendownp  
pendown?
```

Outputs 1 if the pen is down, 0 otherwise

show pendown?

penmode

Outputs PAINT, ERASE or REVERSE

show penmode

pencolor

pc

Outputs the current pen color. This will be a CSS color string, not necessarily the value passed in.

show pencolor

palette *colnumber*

Outputs the a palette entry. This will be a CSS color string, not necessarily the value passed in.

show palette 8

pensize

Outputs a two element list with the pen width and height (usually the same).

show pensize

background

bg

getscreencolor

getsc

Outputs the background color. This will be a CSS color string, not necessarily the value passed in.

show background

6.7 Bitmap Operations

bitcut *width height*

Copy a bitmap, from the turtle's position *width* by *height* pixels.

bitpaste

Paste the previously copied bitmap at the current turtle position.

repeat 4 [rt 90 fd 40] bitcut 50 50 pu fd 100 bitpaste

6.8 Mouse/Touch Queries

mousepos

Outputs a list of the x, y coordinates of the last mouse position

forever [setpos mousepos]

clickpos

Outputs a list of the x, y coordinates of the last mouse press

forever [setpos clickpos]

buttonp

button?

Outputs 1 if any mouse button is down, 0 otherwise.

forever [ifelse button? [pd] [pu] setpos mousepos]

button

Outputs a number indicating the pressed mouse buttons (1 = left, 2 = right, etc) or 0 for none.

`touches`

Outputs a list of current touch coordinates

`forever [ifelse count touches [setpos first touches pendown] [penup]]`

7. Workspace Management

7.1 Procedure Definition

`to procname inputs ... statements ... end`

Define a new named procedure. Inputs can be:

- Required: `:a :b`
- Optional (with default values): `[:c 5] [:d 7]`
- Rest (remaining inputs as a list): `[:r]`
- Default number of inputs: `3`

`to star :n repeat 5 [fd :n rt 144] end`

`define procname [[inputs ...][statements ...]]`

Define a new named procedure with optional inputs

`define "star [[n][repeat 5 [fd :n rt 144]]]`

`def procname`

Outputs the definition of a named procedure as a string

`show def "star`

`text procname`

Outputs the definition of a named procedure as a list, suitable for use with `DEFINE`

`show text "star`

`copydef newname oldname`

Copy a procedure. If a procedure `newname` already existed it will be overridden. Primitive procedures can't be overridden unless `REDEFP` is `TRUE`.

`copydef "new "old`

7.2 Variable Definition

`make varname expr`

Update a variable or define a new global variable. The variable name must be quoted

`make "myvar 5`

`name expr varname`

Like `make` but with the inputs reversed

`name 5 "myvar`

`local varname`

`(local varname ...)`

A subsequent `make` will create the variable(s) in the local scope instead of the global scope

`local "myvar`

`localmake varname expr`

Define a variable in the local scope (shortcut for `local` then `make`)

`localmake "myvar 5`

`thing varname`

Outputs the value of variable. `:foo` is a shortcut for `thing "foo`

`show thing "myvar`

`global varname`

Reserve the variable at the global scope. This doesn't do anything useful.

`global "myvar`

7.3 Property Lists

`pprop plistname propname value`

Set the property *propname* in the property list *plistname* to value *value*.

`gprop plistname propname`

Get the value of the property *propname* in the property list *plistname*, or the empty list if no such property.

`remprop plistname propname`

Remove the property *propname* in the property list *plistname*.

`plist plistname`

Return a list of properties in the property list *plistname*, alternating property name, property value.

7.4 Workspace Predicates

Predicates return 1 (true) or 0 (false)

`procedurep name`

`procedure? name`

Test if there is a procedure with the given name.

`primitivep name`

`primitive? name`

Test if there is a built-in procedure with the given name.

`definedp name`

`defined? name`

Test if there is a user-defined procedure with the given name.

`namep name`

`name? name`

Test if there is a variable with the given name.

`plistp name`

`plist? name`

Test if there is a property list with the given name.

7.5 Workspace Queries

`contents`

Outputs a list with three members. The first is a list of user-defined procedure names. The second is a list of defined variables. The third is a list of non-empty property list names. Only non-buried procedures, variables, and property lists are included.

`buried`

Outputs a list with three members. The first is a list of user-defined procedure names The second is a list of defined variables. The third is a list of non-empty property list names. Only

buried procedures, variables, and property lists are included.

`procedures`

Outputs a list of user-defined non-buried procedure names.

`primitives`

Outputs a list of primitive non-buried procedure names.

`globals`

Outputs a list of defined non-buried global variables.

`names`

Outputs a list with two members. The first is an empty list. The second is a list of defined non-buried variables.

`plists`

Outputs a list with three members. The first is an empty list. The second is an empty list. The third is a list of non-empty non-buried property list names.

`namelist name`

`namelist namelist`

Return a `contents`-style list with the given variable names.

`pllist pname`

`pllist plnamelist`

Return a `contents`-style list with the given property lists.

`arity procname`

Return a list with the procedure's minimum, default, and maximum number of inputs; maximum is -1 if unlimited.

7.7 Workspace Control

`erase contentslist`

Takes a three member list, where the first is a list of user-defined procedure names to erase, the second is a list of defined variables to erase, the third is a list of property lists to erase.

Primitive procedures can't be erased unless `REDEFP` is `TRUE`.

`erase [[myproc] [myvar] []]`

`erall`

Erase all non-buried user-defined procedures, variables and property lists.

`erps`

Erase all non-buried user-defined procedures.

`erns`

Erase all non-buried variables.

`erpls`

Erase all non-buried property lists.

`ern varname`

`ern varnamelist`

Erase the named variable(s).

`epl pname`

`epl plnamelist`

Erase the named property list(s).

`bury contentslist`

Takes a three member list, where the first is a list of user-defined procedure names to bury, the second is a list of defined variables to bury, the third is a list of property lists to bury.

`buryall`

Bury all user-defined procedures, variables, and property lists.

`buryname varname`

`buryname varnamelist`

Bury the given variable (or list of variables).

`unbury contentslist`

Takes a three member list, where the first is a list of user-defined procedure names to unbury, the second is a list of defined variables to unbury, the third is a list of property lists to unbury.

`unburyall`

Unbury all user-defined procedures, variables, and property lists.

`unburyname varname`

`unburyname varnamelist`

Unbury the given variable (or list of variables).

`buriedp contentslist`

`buried? contentslist`

Return 1 if the first named user-defined procedure, variable, or property list exists and is buried, 0 otherwise.

8. Control Structures

8.1 Control

`run [statements ...]`

Run the specified statements once

`run [fd 100 rt 90]`

`runresult [statements ...]`

Run the specified statements once. If the statements return a value, the result is a list with the value as a single member. Otherwise, the result is an empty list.

`repeat expr [statements ...]`

Repeat statements `expr` times

`repeat 4 [fd 100 rt 90]`

`forever [statements ...]`

Repeat statements forever. Used inside a user-defined procedure that terminates with

`output`, `stop` or `bye`

`forever [make "n random 100 show :n if :n == 0 [bye]]`

`repcount`

`#`

Outputs the current iteration number of the current `repeat` or `forever`

`repeat 10 [show repcount]`

`repeat 10 [show #]`

`if expr [statements ...]`

`if [expr] [statements ...]`

Execute statements if the expression is non-zero

`if 2 > 1 [show "yep]`

`ifelse expr [statements ...] [statements ...]`

`ifelse [expr] [statements ...] [statements ...]`

Execute first set of statements if the expression is non-zero, otherwise execute the second set

```
ifelse 1 > 2 [ show "yep ] [ show "nope ]
```

```
test expr
```

```
test [expr]
```

Test the specified expression, save the result in the local scope for subsequent use by `iftrue`

or `iffalse`

```
iftrue [ statements ... ]
```

```
ift [ statements ... ]
```

```
iffalse [ statements ... ]
```

```
iff [ statements ... ]
```

Run the statements if the result of the last local `test` was non-zero (true) or zero (false) respectively.

```
test 1 > 2 iftrue [ show "yep ] iffalse [ show "nope ]
```

```
stop
```

End the running procedure with no output value.

```
output expr
```

```
op expr
```

End the running procedure and output the specified value.

```
catch tag instructionlist
```

Run instructions, but if an error with matching tag is thrown, return the thrown value (if any).

Use `"ERROR` to catch errors from regular procedures.

```
catch "t [ show "before throw "t show "after ]
```

```
catch "error [ show 1 / 0 ] show error
```

```
throw tag
```

```
(throw tag value)
```

Throw an error with the given tag which may be caught. An optional return value can be passed.

```
show catch "t [ show "hello (throw "t "world) ]
```

```
error
```

Outputs a list describing the last error caught: an error number, an error message message, and the procedure name where the error occurred.

```
catch "error [ show 1 / 0 ] show error
```

```
wait time
```

Pauses execution. *time* is in 60ths of a second.

```
bye
```

Terminate the program

```
.maybeoutput expr
```

Like `output` if *expr* returns a value, like `stop` otherwise

```
ignore expr
```

Evaluate and ignore results of the expression

```
make "q [ 1 2 3 ] ignore dequeue "q
```

```
` list
```

Outputs the list with substitutions:

- `, instructionlist` replaced by output
- `,@ instructionlist` replaced by output list members
- `" ,instruction` replaced by output prefixed with "
- `,@instruction` replaced by output prefixed with :

```
show `[a b ,[bf [c d e]] f ,@[bf [g h i]]]
```

```
make "v "x show `[",:v :,v]
```

```
for controllist [ statements ... ]
```

Typical `for` loop. The *controllist* specifies three or four members: the local *varname*, *start* value, *limit* value, and optional *step* size.

```
for [ a 1 10 ] [ show :a ]
```

```
for [ a 0 20 2 ] [ show :a ]
```

```
dotimes [ varname times ] [ statements ... ]
```

Run the statements the specified number of *times*. The variable *varname* is set to the current iteration number.

```
dotimes [ i 5 ] [ show :i * :i ]
```

```
do.while [ statements ... ] expr
```

```
do.while [ statements ... ] [ expr ]
```

Runs the specified statements at least once, and repeats while the expression is non-zero (true).

```
do.while [ make "a random 10 show :a ] :a < 8
```

```
while expr [ statements ... ]
```

```
while [ expr ] [ statements ... ]
```

Runs the specified statements only while the expression remains non-zero (true).

```
while (random 2) = 0 [ show "zero ] show "one
```

```
do.until [ statements ... ] expr
```

```
do.until [ statements ... ] [ expr ]
```

Runs the specified statements at least once, and repeats while the expression is zero (false).

```
do.until [ make "a random 10 show :a ] :a < 8
```

```
until expr [ statements ... ]
```

```
until [ expr ] [ statements ... ]
```

Runs the specified statements only while the expression remains zero (false).

```
until (random 2) = 0 [ show "one ] show "zero
```

```
case value [ clauses ... ]
```

For each clause in order: If the clause is of the form [ELSE *expr*] then *expr* is evaluated and returned. Otherwise, if the clause is of the form [[*matches*] *expr*] and *value* is a member of *matches* then *expr* is evaluated and returned.

```
show case :var [ [ ["a"] "AAA ] [ ["b"] "BBB ] [ else "other ] ]
```

```
cond [ clauses ... ]
```

For each clause in order: If the clause is of the form [ELSE *expr*] then *expr* is evaluated and returned. Otherwise, if the clause is of the form [[*expr1*] *expr2*] and *expr1* evaluates to non-zero (true) then *expr2* is evaluated and returned.

```
show cond [ [ [:var = 1] "one ] [ [:var = 2] "two ] [ else "other ] ]
```

8.2 Template-based Iteration

These higher-level procedures support only the *named procedure* form of template. The first input is the name of a procedure to call.

```
apply procname list
```

Call *procname* with the members of *list* as inputs

```
invoke procname input1
```

```
(invoke procname input1 ...)
```

Call *procname* with the specified inputs as inputs

```
invoke "show "hello
```

```
foreach list procname
```

call *procname* for each item in the list

```
foreach [1 2 3] "show
```

```
map procname list
```

Outputs a list composed of the results of calling *procname* for each item in the list

```
to double :n output :n * 2 end show map "double [1 2 3]
```

```
filter procname list
```

Outputs a list composed of the input list where *procname* called on the item returns non-zero (true)

```
to oddp :n output bitand :n 1 end show filter "oddp [ 1 2 3 4 5 ]
```

```
find procname list
```

Outputs the first item in *list* for which calling *procname* on it returns non-zero (true). If not found, the empty list is returned.

```
to oddp :n output bitand :n 1 end show find "oddp [ 2 8 4 7 10 9 ]
```

```
reduce procname list
```

```
(reduce procname list initial)
```

Call *procname* repeatedly with two inputs - the current value and the next list item. If *initial* is not specified, the first list element is used instead.

```
show reduce "sum [ 1 2 3 4 5 ]
```

```
crossmap procname listlist
```

```
(crossmap procname list list ...)
```

Call *procname* repeatedly with inputs from the lists, in all possible combinations.

```
show crossmap "word [[a b] [x y]]
```

```
show (crossmap "word [a b c] [1 2 3 4])
```

Revision #2

Created 2026-01-13 19:42:45 UTC by Nicolas Farrie

Updated 2026-02-14 13:10:20 UTC by Nicolas Farrie